現場発　状態遷移設計
ZIPC 経由　表面実装生産ライン行き

三洋ハイテクノロジー株式会社
ＳＭＴ事業部　第１技術部　主任

岡本　学

1. Wondrous stories

ZIPC ユーザのみなさんの多くは、小規模組込み機器の開発設計での設計効率を改善するためにツールを検討され、ZIPC が効果を上げることを期待して導入に至ったというような方々なのではないでしょうか。

私たちの辿った道程と現在の状況のお話は、多くの ZIPC ユーザの方々とは全く違った展開で ZIPC と出会い、別の角度から ZIPC に期待を寄せている、という「ちょっと変わった」お話です。

2. In the beginning

私たちは表面実装基板組み立て装置、いわゆる「チップマウンター」を開発／設計しています。実際にご覧になっている方もいらっしゃると思いますが、I/O 点数は非常に多く、制御軸数の多さやそれらの同時制御などを含めて、同期性／非同期性の多くの動作モードが併在するアプリケーションです。表面実装技術普及の最初期では、リード線を電子回路基板のスルーホールに挿入する組み立て方法に比べ、表面実装方法そのものの優位性で組み立て装置の性能を語ることができました。私が入社した 1987 年あたりから表面実装技術が当たり前の技術になりつつあり、それに伴って組み立て装置に求められる性能／機能が高度化してきたため、制御コントローラも 1980 年代前半ではまだ主流だった１ビットマイコン（ICU）に見切りをつけ、８ビット／１６ビット CPU を採用するようになりました。しかし、その当時はまだ「制御ソフトウェアの生産性」はアセンブラ／コンパイラを走らせるワークステーションの性能やデバッグで使用する ICE の能力の問題として見られており、「生産性」そのものを本当の意味で理解している人はいませんでした。時代が昭和から平成に切り替わった頃のことだったと思います。私を含め数名が社内のシステムエンジニアリング関連のセミナーを受講し、構造化分析に代表される開発工程や開発技法といった概念を持ち帰りました。確かにそこ

で見たHatley／Pirbhaiらの「リアルタイムシステムの構造化分析」は論理的であり、開発の実態の暗黒部分に光を照らすようにも思えました。なんとかこれを実際の開発／設計業務に生かせないかと関連する書籍や事例発表論文などをあたってみましたが、その初期の頃に感じた「実際の業務にこのまま取り入れても普及させられないし、当然効果も期待できない」という直感が確信に変わるばかりでした。欲しかったのは理論ではなく、確かに何かを生み出すことのできる「手法」だったのです。

3. Knife edge

我々の制御ソフトウェアの開発の現場でも、一般的に問題点として指摘されるような事例を山ほど抱えていました。中でも深刻だったのは制御ソフトウェア設計に関するドキュメントが全くと言っていいほど機能していない点でした。制御アプリケーションの機能仕様の面では「暗黙の仕様」という形でドキュメント化されていないと見ることができる部分もありましたが、構造的な部分の仕様に関しては「何も無い」か「あっても役に立たないフローチャートしかない」という状態でした。その頃の私は、一般自動機の制御を一時担当した後、画像処理アプリケーションのソフト開発を担当していました。その一方で開発ツール整備のプロジェクトにも関わり、制御アプリケーションの問題も間近に見えるようになりました。画像処理アプリケーションでは処理構造をフローチャート的に捉えて仕様として表現することができましたが、制御アプリケーションではフローチャート的に書かれた部分がシステムの「ガン」になるという実例がいくつもありました。制御アプリケーションの中で比較的うまく行っている部分をピックアップすると、状態変数に現在の状態を記録しswitch-caseによる処理分岐を使うという構造も浮かび上がってきました。今思えばごく原始的なステートマシンの実現方法ですが、この当時私自身も「ステートマシン」という言葉は全く知りませんでした。こういった状況を自分なりに分析した結果、私は2つの結論を出しました。システムをうまく動かすためには状態変数を使って制御アプリケーションの構造を表現しなければならないということ、そして、自分が制御ソフト開発チームに異動し実際に制御アプリケーションを設計してみなければならないということ、これを今は故人である当時の私の上司に直訴しました。1993年のことだったと思います。

4. Armed and ready

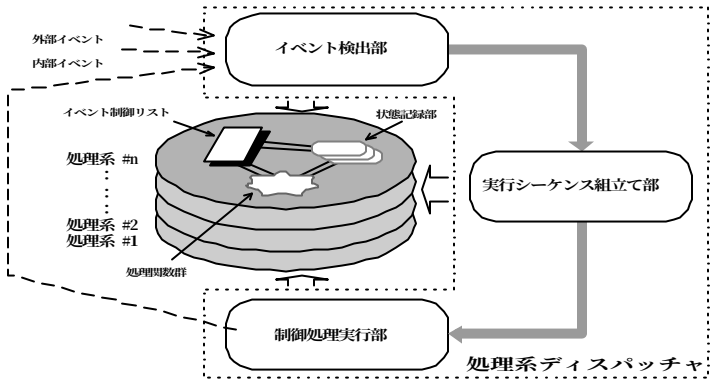
その当時担当業務が変わることについて、私にどのような役割が期待され、開発チームリーダ間でどのような駆け引き（笑）があったか今となっては全く分かりませんが、ともかく私は件の上司が体調を崩して開発チームから外れるのと同時に、たった一人の部下として「制御ソフトウェアの標準化推進」という実に胡散臭い業務担当となりました。名目はどうあれ実際には制御ソフト開発チームに混ざってソフト設計を中心に開発に携わることになり、これで活動の準備は整いました。

幸い、制御ソフト開発チームには私の考えに賛同してもらえる人がいました。やはり同じように問題意識を持ちながらも精神論的な設計手法の導入効果にはあまり期待できないという考えを持っていました。そんな何人かで雑談したとき、結果に繋がらない理論はクソの役にも立たない、いくら構造化分析でアウトプットが出てもコード化してプログラム実装するときにまたイチから考え直しでは全く意味が無い、コード化が容易になる仕組みに繋げていかなければならない、という見解で一致したのを覚えています。これで目標は決まりました。実現するためのよりどころとなるものは既に決まっていました。状態変数に現在の状態を記録して制御する処理構造としてコード実装する方法で制御アプリケーションが十分うまく動かせるはずだ、という確信がありました。問題は、われわれのアプリケーション、すなわちチップマウンターの制御では処理系がかなり多くなること、またリアルタイム性が要求される一方、処理系間の同期性も崩してはいけないこと、従来の方法の中でこれらの制約をクリアしてきた処理方法のノウハウをどこまで継承できるかということでした。

長らく机上で構想を練りに練った結果、１つ以上の完全な状態遷移を含む処理系を、テーブル化されたイベント制御リスト、同一処理をタイミングをずらして複数起動可能な状態記録ブロック、実際の制御動作を記述した処理関数群で構成し、検出したイベント等に応じてこれらをディスパッチさせるコントローラのコーディングモデル第１号が完成しました。

日付が1994/3/25とプリントアウトされたリストが当時のファイルに残っています。仕組みを一言で表わすなら、「非プリエンプティブなリアルタイム OS のタスクスケジューラ部分だけ抜き出したもの」でしょう。表現的にはムチャクチャ矛盾していますが、それがこの仕組みの肝心なところと理解していただきましょう。とにかくこれで実行環境も整いました。

（図１：ディスパッチコントローラのモデル図）



5. Born to be wild

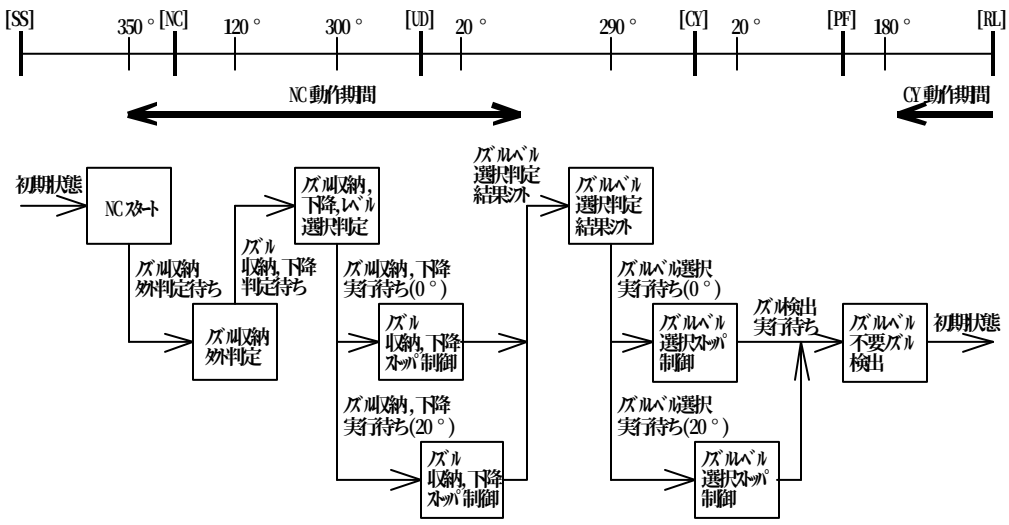
状態遷移で設計していこうと決めた1993年から1994年にかけて、それまでそういう手法に親しんでいないメンバーにどのように状態遷移を考えてもらうのがよいか、ということも考えていました。チップマウンターのような機械の制御の中で最も重要な設計情報はタイミングチャートです。そこで、タイミングチャートのバリエーションとして状態遷移を記述する方法を考えました。すなわち、横軸が時間経過となり同期性／非同期性を含めた形で表現したイベントを配置し、イベントに関連付けられたアクションがボックスとして、処理系の継続する状態が矢印として表わされ、ちょうど一般的な状態遷移図とはボックスと矢印の役割が逆のような形になりました。状態遷移を記述する場合の補助となるようなツールが無いかどうかも調べました。CASEツールの中で状態遷移図が描けるものもありましたが、その当時いずれも高価なツールのパッケージの中のホンのオマケ程度のものだったり、安価なものも見るからに機動力に欠けるものだったりしました。この頃UNIX上で走る状態遷移表エディタを見た覚えがありますが、状態遷移図を中心に進めたかった上、表記方法（特にイベント）にもイマイチ自由度が無かったりしたので資料請求もしませんでした。後で聞いた話では、どうもこのとき見たのがZIPCの初期モデルだったようですが、真相が明らかになることは無いでしょう。結局、表記の自由度と

ワープロ文書内への埋め込みを考えるとドロー系ソフトしか選択肢がありませんでした。タイミングチャート兼用型というオリジナル技を編み出してしまったこともあって、状態遷移図エディタは長年の懸案となっており、今もってなお満足できるものに出会えていません。

6. Carry on wayward son

やがて開発が進むにつれ、実際のソフト設計担当者の間で「このやり方は結構いいぞ」という感触が出てきました。相変わらず補助となるツールは状態遷移図を描くときのドロー系ソフトだけでしたが、それもあまり気にならないくらいまで「慣れ」が出てきました。デバッグのスタイルも変わりました。機械制御では、プログラムをブレークポイントで止めてステップ実行で確認、というテクニックは使えません。状態遷移方式でないときはソースコードを見比べながら関係ありそうな部分のトレースにヒットするまで片っ端からトリガをかけまくる、というケースもままありましたが、状態遷移方式では異常動作の近辺での遷移の辻褄が合っているかをまず検討し、それを手がかりに絞り込んでいくことが容易になりました。全ての処理系の遷移の履歴はディスパッチャによってログとして残すことができるため、再現頻度の低い現象も捕まえやすくなりました。しかも多くのケースは遷移を検討する段階で原因が見つかります。複雑な処理系や、制御タイミング的にさらに厳しいシステム要求、

（図２：タイミングチャート兼用型状態遷移図の例）



またいくつか形態の異なる機械の制御への適用などを経て改良／チューニングを重ねていますが、途中最も大きな変化はリアルタイムOSとの融合でしょう。処理系のディスパッチもタスクとしてリアルタイム OS にやらせればいいようにも思えるかもしれませんが、それがそう甘い話ではない…という話は長くなるので省略させていただきましょう。とにかくOSは主にイベント検出部に入るまでのイベントの通り道をすっきりさせる目的で使われていて、今も1994年の原型とそう変わらない形で処理系ディスパッチの仕組みが制御動作実行タスクとして働きつづけています。

7. Now I'm here

1998 年にスタートした新モデルのチップマウンター開発プロジェクトでは、装置内部の制御システム構成の一新に合わせて状態遷移設計手法の適用範囲も広げました。これまで通りディスパッチャが実行する部分の記述の他、リアルタイム OS がタスクとして直接実行する部分の記述、制御動作処理以外のステート記述についても状態遷移を取り入れました。当然、制御動作の実現で成果が認められ、状態遷移設計手法がある程度浸透してきている、という背景があります。

さてこうなると状態遷移の表現はタイムチャート方式だけではなくなりました。ちょうどプロジェクトがスタートした頃、組込み開発関連の展示会で PC 上できびきび動く ZIPC とやらを目にしていました。そこでやっと本格的にそいつの導入を検討し、最終的に購入することになりました。導入の決め手は、ツールがモジュール化されていて必要な部分をスポット的に使うことができそうだ、という点でした。こんなことを口走ると開発元であるキャッツさんには申し訳ありませんが、私たちは開発工程の多くをZIPCに任せてしまうつもりはありません。スタイルに合ったツールを選ぶ、無ければ自分たちでどうにかする、という考えで今後も進みたいと思っています。

	制御処理系動作	OSタスク	動作処理以外のステート制御
実行環境	専用デバスパッチャあり	OSシステムコールを利用	専用のイベント検出部あり
設　計	ドロー系ソフト 一部ZIPCエディタ	特になし★	ZIPCエディタ
モデルシュミレーション	特になし	特になし	特になし
コーディング	テーブル化による コードモデル	特になし	特になし★
コードシミュレーション	自社ツール	特になし	自社ツール

★ :本文によるコメント参照

状態遷移設計手法をベースにした実行環境は構築しましたが、そのために設計者に特定のツールの使用を強制するようなことはなるべく避けています。ですから、ZIPCとの付き合い方も「使えそうな部分を選んで自分たちのやり方に当てはめる」という形になります。現在私たちが実際にZIPCをどのように使っているかといえば、もう99%はエディタのみです。状態遷移表の書き方に不慣れな期間はチェッカもたまに使っていましたが、今はほとんど使っている人を見かけません。状態遷移設計手法の適用範囲とそれぞれの工程でよく使われるツールをまとめると、だいたい次のような感じになっています。シミュレーションに関してはイベントを仮想的に与えてコーディング後の動作を確認できる自社ツールを開発しており、モデルシミュレーションとなるZIPCシミュレータに関しては差し迫った必要性は感じていません。特に制御処理系動作に関して、今のレベルでは、テーブル化されたパターンに沿ってコーディングモデルを作る負荷に比べZIPCシミュレータを実行させるために必要な環境設定の負荷のほうが大きく、変更フィードバックサイクル的にも無理があるためです。だからと言ってZIPCに多くを期待しないわけでもありません。導入当初からコードジェネレータの可能性が念頭にあり、初めは制御処理系動作のコード化に使えるかどうかを考えていましたが、しばらく使い方の様子を眺めたり、実際にジェネレータでコード化したものを見たりした結果、動作処理以外のステート制御のコード化のほうが労力的にも設計品質的にも導入効果が高そうだ、と感じています。実行環境としてオリジナルのイベント検出部を持っているため、そことの組み合わせでもう少し手間がかからなくなれば利用価値に疑問の余地はありません。また、現状全くの真空地帯となっているOSタスクの設計に関してもZIPCが効果を発揮してくれることを期待しています。アプリケーションの全体構成からするとOSタスクになりうる部分の重要度はそれほど高くなっていないことが多いのですが、それが逆に設計ドキュメントの不足という形で現われやすくなっており、他の適用分野での実績と信頼を引っ提げてここを制するようになってほしいと思っています。

8. Meek shall inherit nothing

既にZIPCを導入しているユーザさんには今更意味無いアドバイスかもしれませんが、状態遷移設計手法というのは設計者が理論を理解していなくても工夫次第で受け入れてもらえるものです。開発ツール整備を担当する方は「そのために

ZIPCを見せ玉に使う」くらいに考えると気が楽になるのではないでしょうか。
手法そのものに対する「慣れ」が出てくると、ツールへの依存度が減ってきます。使わなくても済むものは使わなくなるし、ツールを使って解決したいポイントが別の機能に移っていったりします。ツールが提供する機能が高度であればあるほど、そういう傾向があると思います。そういう意味で、ツールへの「慣れ」が上回っている設計者には注意が必要です。使えと言われてるのでただの表を書くのにZIPCエディタを使っている、というのでは本当に使っているとは言えません。状態遷移を書くのと同じ頭で直接コーディングができるようになってくれたほうがマシでしょう。そのために設計ドキュメントが残らなくなってしまうのは本意ではありませんが。(笑)

9. Great expectations

一貫した開発ソリューションとしてユーザがZIPCに期待し、ベンダとして先進的な機能を備えてそれに応えていこうとする方向は決して間違っていないと思います。私が述べたような方向性を指向するのは少数派ではなくても非主流になるかもしれません。しかも独自の実行環境に対してスポット的なツール利用を考えるユーザなどニッチの極みとも言えましょう。しかし、私たちに都合のいい言い方をさせていただくと、そんなユーザを同時に満足させるところにこそ ZIPC の存在が光を放ち続ける理由があるのではないでしょうか。全能の神にこそ、迷える仔羊に手を差し伸べることを忘れて欲しくない、そんな心境でもあります。
ん？私たちが仔羊？いや、そんなかわいらしいもののはずがないですね。